Automating The Design of Specification Interpreters

Kurt Stirewalt, Spencer Rugaber, Gregory Abowd

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Abstract

In this paper, we demonstrate the use of model checking in an automated technique to verify the operationalization of a declarative specification language. We refer to an *interpreter synthesizer* as a software tool that transforms a declarative specification into an executable interpreter. Iterative approaches to synthesizer generation refine initial synthesizer designs by validating them over a test suite of specifications. Carefully chosen test suites and structural constraints enable inductive reasoning with support from a model checker to assert the correctness of generated interpreters. This iterative approach to synthesizer generation occurred naturally in our work on developing interpreters for declarative human-computer dialogue languages as part of the DARPA MASTERMIND project. We will discuss the issues underlying the translation, operationalization and verification of the hierarchical task language for MASTERMIND. We will also discuss the importance of this semi-automated, iterative approach for assessing non-functional design tradeoffs.

Keywords: model checking, declarative specification languages, model-based user interfaces, automated verification

1 Introduction

Reactive, or event-driven, software systems are difficult to build and maintain. A good example of such an event-driven system is the graphical user interface, in which the behavior of the system is driven by user activity. A declarative specification language can ease the initial description of such a reactive system, and in the case of graphical user interfaces, the specification language is referred to as dialogue, or task, language [4]. In the MASTERMIND project, we are concerned with the development of tools to automate the generation of graphical user interfaces based on declarative task specifications. Our approach is similar to that taken in the model-based user interface community [11].

One problem that we encountered is the difficulty of formally proving the correctness of an interpreter built for the task specification language. This is a general problem of *operationalizing* a declarative specification language. Operationalization is the process of generating an *interpreter synthesizer* for a declarative specification language. The synthesizer generates a reactive state-based interpreter to implement the specified behavior. Operationalized specification languages enable engineers to specify and reason over systems and then feed the specifications into a compiler that generates an executable simulator.

Operationalization as a process is inherently an engineering activity. Analysts design synthesizers that satisfy interdependent constraints from two major sources: specification language semantics and customer efficiency requirements. Variations in the specification language imply changing semantic constraints on the design of the synthesizer. Likewise, requirements on the synthesized interpreters manifest themselves as constraints on synthesizer design. An engineering process for operationalization must support the generation of synthesizers that simultaneously satisfy constraints from both sources. We

found symbolic model checking technology useful in automating this simultaneous solution process. The results we present in this paper directly address the translation from a specification language—in our case the MASTERMIND task modeling language—to an executable interpreter consisting of communicating state machines. We describe an approach that addresses the treatment of non-functional efficiency requirements. The end result of the design process is a set of tools to generate and verify a synthesizer and partial results that could be used to reapply the process in the face of change (e.g., if the language changes, or the efficiency requirements of the customer change.)

Overview of paper

In Section 2, we relate this application of model checking to other work on applying model checking to software systems. Using the specific example of the MASTERMIND task modeling language, we explore in Section 3 the issues surrounding the introduction of automation into the engineering process of generating an interpreter from a declarative specification language. The approach we use is to translate the language into an interpreter using concurrent state machines. There are then trade-off issues dealing with the mechanics of how communication occurs between these state machines. In order to follow an engineering approach to examining these alternatives, we need to invoke some automated support, and this is where we employ modern model checking technology. In Section 4, we discuss how we use a particular model checking technology, the Symbolic Model Verifier (SMV)[9]. Using this tool we were able to experiment with design tradeoffs to ensure the product meets its functional requirements, and to quickly understand how inconsistency arises. Additionally, the infrastructure facilitates tuning a design to non–functional user requirements without sacrificing correctness. In Section 5, we outline a procedure for this automated trade-off analysis.

2 Related Work

There has been increasing attention given by the software engineering community to the merits of model checking. On the surface, the approach is very simple. A problem is described in terms of a state-based model description and that model is exhaustively searched in order to verify that it upholds certain properties. The cause for the recent surge in interest can be linked to developments that have provided for efficient search across very large finite state spaces using symbolic instead of explicit means of representation [3, 9]. The majority of the work in model checking has been demonstrated on hardware problems, because they are better suited to model checking technology's restriction to finite models. Applications of model checking to event-based software systems which have a finite state representation have been demonstrated for examining timing requirements [2] and for verifying properties of production rule dialogues in a human-computer interface [1].

Model checking is good at pointing out errors in a specification, but it requires a finite state representation of a problem. Wing and Vaziri-Farahani [12] provide an elegant argument justifying the application of model checking technology to software problems with infinite state spaces through finite abstractions. Jackson [8] used model checking technology to efficiently search a finite pruning of a Z specification state space to detect errors. We are also taking advantage of the error spotting in our iterative approach to verifying a synthesizer generator in this paper. Our work here is a unique application of model checking to the problem of matching a declarative language to an automatically generated interpreter. Our tools not only employ a symbolic model checker to do exhaustive search, as done by all of the approaches listed above, but they also generate the properties of specifications that need to be checked.

3 Case Study: The MASTERMIND Task Language

A reactive system specification environment provides designers with a language for specifying reactive systems, a suite of tools for reasoning over specifications, and a compiler that generates executable run–time systems from specifications. Such environments differ in the nature of the specification language, the degree of support for reasoning, and the extent to which the generation of run–time systems is fully automated. MASTERMIND is an environment that specializes in generating highly interactive user interfaces and attaching them to applications [10]. The MASTERMIND project involves a number of different models that are relevant for interactive systems design (presentation, application, and task) and we are focusing here on the task model and its associated specification language. A suite of design critics help designers reason about these specifications, and the task interpreter synthesizer generates executable run–time interface drivers from these specifications. The MASTERMIND approach is useful because its task specification language directly supports hierarchical task analysis. Features of the language complicated the design of the interpreter synthesizer. This section explores those complications.

3.1 The Language

Research in human–computer interaction suggests that hierarchical task analysis is a natural means of dialogue specification [4]. The MASTERMIND task language is a visual language for expressing the results of such analysis. It allows designers to specify task hierarchies, hierarchical ordering constraints, and interaction with an external environment in the form of preconditions and effects. The visual language has a textual analogue (which we will use in this paper) containing only hierarchy and ordering information.

An example helps motivate the language. Suppose we are to design the interface for a simple e–mail program. Users of this program want to accomplish two main tasks: sending mail and reading mail. In the designer's mind, users perform one of these tasks or the other, but not both at the same time. This is expressed in the task language by declaring the *Mail* task to decompose alternatively into the *Send_Mail* task and the *Read_Mail* task:

$$Mail ::= \textbf{alt}(Send\_Mail, \; Read\_Mail)$$

The keyword **alt** specifies that one and only one of the subtasks may be performed. The designer must now refine the two subtasks. The *Send_Mail* task has three major subtasks: enter the name of the recipient, compose the message to be sent, and send the message. The designer rationalizes that these subtasks must be performed in that order and declares the following:

$$Send\_Mail ::= \textbf{seq}(Enter\_Recipient, \; Compose\_Message, \; Press\_Send)$$

Each of these subtasks must be refined. The simplest is *Press_Send* which represents an atomic user interaction via a mouse press. It has no subtasks, and so is declared as a **leaf** task:

$$Press\_Send ::= \textbf{leaf}$$

In addition to those already described, the language provides the following ordering operators:

par specifies that subtasks may be performed in any order and interleaved. For example, the *Enter_Recipient* and *Compose_Message* tasks might be ordered in this fashion rather than sequentially, allowing users to perform parts of both both tasks in any order.

opt specifies that a single subtask may execute, or may be skipped if the user chooses to perform a subsequent subtask. An **opt** task in the e–mail example might be the a task to enter Carbon copy (Cc) recipients.

cond specifies that a single subtask will execute in the situation where an environmental condition is initially satisfied. If an e–mail user imports

non–ASCII data in the text of the message, the system could issue a warning to inform this person that some mailers might have difficulty with the message. A **cond** task could be used to place the warning at an appropriate point relative to other tasks.

continuous specifies that a subtask executes continuously as long as some environmental condition is satisfied. The *Read_Mail* task, for example, might reissue the read prompt as long as there are unread messages in the mail box.

repeat specifies that a single subtask executes repeatedly as long as the user chooses to request it. Rather than choosing to return the user to the read prompt based on the mail box contents, the *Read_Mail* task could instead allow the user to explicitly repeat the task or continue with some subsequent task.

The **opt**, **cond**, **continuous**, and **repeat** ordering operators are required to operate over a single subtask. The **seq**, **par**, and **alt** operators operate over any finite set of subtasks.

3.2 State Machine Interpretation

Since the task language expresses hierarchical ordering invariants over subtasks, its operationalization was difficult. The first issue we faced was deciding the representation of the interpreter. Task specifications express the legal orderings of user and system interactions. Operational interpretations must actively monitor interaction and constrain the system so that only legal orderings may occur. Specifically, if in some state only a certain set of interactions is legal, the interpreter enables only the widgets associated with this set of interactions. Enabling a widget means allowing that widget to accept mouse and/or keyboard events. When the system changes state in response to an interaction, the set of legal leaf tasks will likely change. The interpreter handles this by disabling enabled widgets not in the new set and enabling those in the new set that were not in the old set. This is typically indicated by greying out buttons, menu items, etc. The legality of a given interaction changes depending upon temporal context.

If, for example, we declare that the tasks **Enter_Recipient** and **Compose_Message** are to be strictly sequenced, interactions that comprise **Enter_Recipient** must be enabled for the duration of that task but disabled when **Compose_Message** begins. Our language is such that the legal interactions associated with a state are statically determinable. Given this behavior it seems natural to associate sets of legal interactions with states in the generated interpreter.

3.3 Correctness by Induction

Engineers are obliged to show that synthesizer designs yield a generator that produces correct interpreters. Deductive proofs of correctness are ideal, but they are difficult to construct. If some aspect of the problem can be shown to have inductive structure, then we could exhaustively show correctness on a small collection of tests and use the induction principle to assert correctness on any specification. An inductive proof of design correctness utilizes a technique called *structural induction*. Structural induction establishes the correctness of a synthesizer over a set of atomic tasks (the base case) and then argues that simple compositions of these tasks are correct (the inductive step). The induction principle then says that the synthesizer generates a consistent machine for specifications of any size.

We developed a test suite of example specifications that exhaustively exercises both the base case and the inductive step. If a synthesizer passes the test suite, we assert semantic confidence in that product. For the proof to hold, we must argue that specifications not in the set behave like compositions of specifications in the set (the structural induction argument).

Specifically, the synthesizer must generate interpreter components that communicate hierarchically because the ordering operators correspond to tasks with hierarchical subtasks. This means that the output representation cannot be a simple state machine. Simple state machines retain compositional structure for some of the task language operators but not all of

them. Composition using the **seq** operator, for example, can be preserved by making the final state of the machine synthesized from the first subtask the start state of the machine generated by the second subtask. The alternative operator is likewise preserved. The **par** operator, however, is not preserved in the structure of the machines. The only general way to implement the **par** of two state machines is to generate a machine whose states are the cross product of the states of the constituent subtasks, yielding an explosion of states. This problem necessitates a different representation for the output of the synthesizer. Since simple state machines cannot deal well with parallelism, we look to a system of concurrent state machines [6].

3.4 Concurrent Mealy Machines

We chose to make the synthesizer translate each task into a Concurrent Mealy Machine. Mealy Machines[7] are deterministic finite state machines that consume and then produce a symbol at every state transition. Concurrent Mealy Machines treat the produced symbols as events and allow the produced events of one machine to be the consumed events of others. Mealy Machines compose by *synchronous parallel composition* [5] With only seven types of tasks in the language, we need only seven *classes* of Mealy Machine. Each different class of task ordering is captured by a canonical machine that controls the execution of subtasks by issuing control events. Interaction tasks are also associated with machines and respond to **Enable** and **Disable** control events issued by machines upholding ordering invariants.

Mealy Machines enforce ordering invariants by observing the behavior of other machines and issuing control events that enable or disable these machines. Subtask machines must have facilities to respond to control events, and control events must be targetted to particular machines. Since tasks compose hierarchically, any type of task might be the subtask of another task. The following states exist in all machine classes for ordering control:

notready indicates this machine is not able to do anything because of ordering constraints.

ready indicates this machine is able to perform an activity but has not yet begun.

active indicates this machine is presently participating in some form of activity.

With these canonical states there are canonical events that cycle through them.

Enable forces a machine in the **notready** state to transition to the **ready** state.

Disable forces a machine in the **ready** state to transition back to the **notready** state.

3.5 Example Design Decision

Consider modeling part of the dialogue of a World Wide Web browser. Users may move forward one page or back one page by pressing arrow buttons on the toolbar. One cannot, however, do both at once. The tasks associated with this choice are modeled in our formalism as follows:

$$History ::= \textbf{alt}(Move\_Forward, Move\_Back)$$

(where *Move_Forward* and *Move_Back* are interaction tasks). The synthesizer generates a Mealy machine for each of the three tasks. At some point, the *History* machine enables its subtasks by issuing an `Enable` event to both *Move_Forward* and *Move_Back*. Once enabled, widgets associated with these machines are enabled, and the user may physically interact with one of them to communicate a choice. If the user clicks on the widget associated with the *Move_Forward* machine, the *History* machine must issue a `Disable` event to the *Move_Back* machine.

The **alt** machine (*History*) must respond to the activity of subtask machines to know when to issue `Disable` events to others. One way to implement this is to make the ordering machines like **alt** *peek* into the states of subtask machines and transition based on the results of these peeks. Another implementation forces subtasks to *announce* activity to parent tasks. Each approach has advantages, and the decision to use one or the other is a trade–off between time and space requirements. Peeking is the least sensitive to

race conditions, but uses a lot of space for deep hierarchies. The announce option, on the other hand, uses constant space but must make a number of steps to get to its destination. The difference between the two becomes more noticeable with deeper hierarchies:

$$
\begin{aligned}
\mathit{Mail} &::= \mathbf{alt}(\mathit{Send\_Mail}, \\
&\qquad \mathit{Read\_Mail}) \\
\mathit{Send\_Mail} &::= \mathbf{seq}(\mathit{Enter\_Recipient}, \\
&\qquad \mathit{Compose\_Message}, \\
&\qquad \mathit{Press\_Send}) \\
\mathit{Read\_Mail} &::= \mathbf{seq}(\mathit{Select\_Mailbox}, \\
&\qquad \mathit{Select\_Message})
\end{aligned}
$$

The Mail task controls the exclusive choice of interaction tasks *Enter_Recipient* and *Select_Mailbox* which are not its immediate subtasks.

If the machine associated with *Mail* uses peeking to detect activity, it must be able to peek into the internal state of both interaction tasks. There could be many of these interaction tasks. The ordering machine might need to peek into as many as 2^k machines where k is the depth of the hierarchy rooted by the alternative task. Peeking clearly gives up space to achieve more rapid switching.

Instead of peeking, machines could announce activity by issuing an `Activate` event to their immediate parent. The parent would then perform any ordering activity appropriate at its level in the hierarchy and issue another `Activate` event to its parent. In the *Mail* example, when the *Enter_Recipient* machine notices interaction, it issues an `Activate` event to *Send_Mail*. *Send_Mail* will then issue an `Activate` event to *Mail*. At this point, the *Mail* task issues a `Disable` event to *Read_Mail* which in turn sends a `Disable` event to *Select_Mailbox*. Announcing activity clearly introduces delay in terms of propagation to achieve constant space increase.

The mechanism for responding to activity poses a design choice. Engineers can weigh the cost/benefits of time and space usage, but they have no way of knowing the solution is correct. Both approaches seem to work for this small example, but there are six other types of ordering operators. A tool that judges the validity of such design decisions would be beneficial.

4 Model Checking

Which observation mechanism should we use? Both solutions have good and bad properties. The peeking solution is easier to reason about because it subsumes the need to propagate events throughout the hierarchy. Unfortunately, it comes with a possibly exponential cost in terms of added states. The announcement of activity solution, on the other hand, is not as easy to reason about but uses much less space. Ultimately, engineers will decide which strategy to apply and will need to demonstrate the correctness of the choice with respect to the task language. Some of these solutions are difficult to reason about, and decisions tend to be subtly inter–related. If tradeoffs like peek vs. announce come up for each ordering mechanism, decisions made in one case will likely invalidate assumptions made in others. Fortunately, automated tools can be applied to assist in the validation.

In our approach, engineers associate each type of task in the specification language with a unique *class* of state machine and a schema of temporal invariants demonstrating the ordering semantics of the task. A tool instantiates specifications into a collection of concurrent Mealy Machines and a collection of temporal invariants whose composition demonstrates the correctness of the machines. The output of the translation tool can be fed into the SMV Model Checker to be verified. When this step passes for each specicifation in the test suite, the design is correct.

4.1 The SMV Model Checker

The SMV model checker reads the specification of a system of concurrent state machines and a collection of temporal specifications over the states of these machines. SMV state machines are defined in two parts. VAR blocks declare variables that retain internal state. States of the machine are taken to be all combinations of variable values. ASSIGN blocks express transition functions. Initial values are assigned to each of the state variables through the init(...) assignment. The transition function is constructed by associating a state (set of variable values) with a set of possible next states (new values of variables) using

the next(...) state feature.

A MODULE in SMV is a parameterized encapsulation of state machine declarations. MODULES can be instantiated into the state of other MODULES. An unparameterized distinguished MODULE *main* contains the concurrent state machine instantiations that make up a system. During instantiation, MODULE parameters of a machine can be bound to internal state variables of other machines.

4.2 State Machine Conventions

Engineers define classes of Mealy machines corresponding to the different features of the specification language. Since machines must compose hierarchically, we provide a *template* of minimal state functionality that each class must possess. Machine classes take the form of parameterized SMV modules[1]. Machine instantiation then corresponds to MODULE instantiation. A template addresses two needs:

1. support for general hierarchical composition, and
2. augmentation with ordering invariants.

The template reserves the names of two internal state variables: `state` and `event`. The `state` variable maintains the state of the particular machine. Interaction widgets and application service invocations are associated with these states. The `event` variable remembers an event issued to this machine by some other. This is necessary because SMV does not directly support event/broadcast mechanisms. Machines must influence the state of other machines by issuing `events`. A machine may not directly modify the `state` variable of another machine.

For machines to issue events to other machines, they must be connected. Connections represent the minimal state required for two machines to communicate (access each other's internal states and issue events to one another). Connections are directed in the sense that they always match a parent machine to a child machine, but both machines can issue events to each other. To support connection, all classes of machine have the following two MODULE parameters for each machine with which they might need to communicate. One of these parameters is bound to the machine, and the other parameter is bound to the `event` variable of the machine. The binding of the `event` variable of other machines allows this machine to modify the field (issue the other machine an event). Two parameters are necessary because SMV visibility rules only allow modification of variables that are explicitly passed as parameters. Those variables that are brought into scope by the machine binding are effectively read–only.

The hierarchy:

$$
\begin{aligned}
\mathit{History} &::= \mathbf{alt}(\mathit{Move\_Forward}, \\
&\qquad\qquad \mathit{Move\_Back}) \\
\mathit{Move\_Forward} &::= \mathbf{leaf} \\
\mathit{Move\_Back} &::= \mathbf{leaf}
\end{aligned}
$$

generates the following instantiation of machine classes:

```
MODULE main
VAR
   History  : process alt(Move_Forward,
                          Move_Back,
                          Move_Forward.event,
                          Move_Back.event);
   Move_Forward : process leaf(History
                           History.event);
   Move_Back    : process leaf(History,
                           History.event);
```

The machine `History` has a connection with the machine `Move_Forward`. `Move_Forward` is instantiated with `History`'s event variable and so may issue an event to `History`. Likewise, `History` is instantiated with `Move_Forward`'s event variable.

4.3 Invariant Conventions

To validate the instantiated machines we must also instantiate the invariants that express their correctness. SMV expresses state machine invariants using Computation Tree Logic (CTL). CTL is a subset of branching time temporal logic that allows the specification of temporal properties over paths execution. SMV uses model checking to verify or refute CTL formulae over classes of concurrent state machines.

[1] To Referee: We have included two such classes in the appendix. The classes for the other ordering operators can be included if you think they would be instructive. We did not include them for reasons of space.

When the CTL expression is of a certain form, SMV can actually construct a counter–example to demonstrate a failure case. To the extent possible, we exploit that form in the invariants we instantiate so that engineers will get constructive feedback when their designs are flawed.

The semantics of the various task language features are captured in a quantified tree language that facilitates instantiation alongside the state machines. In general the ordering operators demonstrate semantics by proving two types of properties:

safety which demonstrate that the orderings are not violated, and

liveness which demonstrate that a maximal number of legal choices are presented to the user.

We take as an example the semantics of the **alt** ordering operator. Alternative ordering explicitly prohibits concurrent activity between machines in disjoint subtasks. The disjointness can be captured by the CTL expression:

```
AG !((t1.state = active) & (t2.state = active))
```

where `t1` and `t2` are machines associated with subtasks at some level from the two disjoint subtasks. The semantics of alternatives imply that the user *can* make a choice. This liveness property is captured with the following CTL expression:

```
EF ((t1.state = ready) & (t2.state = ready))
```

where `t1` and `t2` are machines associated with the direct subtasks of the particular task.

Interestingly, liveness properties tend to be adequately captured by temporal constraints over immediate subtasks; whereas safety properties tend to require temporal constraints over all possible pairs of descendant subtasks.

4.4 Announce Example

The MODULE leaf (shown in Figure 1) represents an interaction task. Needing to communicate only with its parent in the hierarchy, it takes two parameters. From the `ready` state, the machine could receive an `Activate` event. Since parent tasks assume subtasks notify them of activity, this task must issue an

```
MODULE leaf(parent,parevent)
VAR
   event    : {Null, Disable, Enable, Activate};
   state    : {notready, ready, active};
ASSIGN
   init(state) := notready;
   init(event) := Null;
   next(state) :=
        case
            (state = notready) &
            (event = Enable)    : ready;
            (state = ready) &
            (event = Disable)   : notready;
            (state = ready) &
            (event = Activate)  : active;
            (state = active)    : notready;
            1                         : state;
        esac;
   next(event) :=
        case
            state = ready        : {Activate, Null};
            1                       : Null;
        esac;
   next(parevent) :=
        case
            next(state) = active : Activate;
            1                    : parevent;
        esac;
```

Figure 1: SMV description of a leaf interaction task.

`Activate` event to its parent. This is achieved through the assignment `next(parevent) := ....` Each class of machine has similar functionality to support the propagation of `Activate` events up the hierarchy. MODULEs representing the **alt** and **seq** ordering machines appear in the appendix.

Unfortunately, when we apply SMV to the machine instantiation and invariant instantiations of the *History* hierarchy, the invariant does not hold. SMV provides us with the counterexample of Figure 2. The inconsistency arises from an event propagation race condition. The counterexample shows an execution sequence in which both `Move_Forward` and `Move_Back` become active before `History` is scheduled to run. Perhaps machines should check for sibling activity rather than relying on control events to arrive. This observation led us to the peeking solution.

4.5 Peek Example

The next design makes use of the DEFINE primitive in SMV which allows designers to associate a symbolic name with a formula over system state. These macros

```
state 1.1:
Move_Forward.event = Enable
Move_Forward.state = notready
Move_Back.event = Enable
Move_Back.state = notready
History.event = Disable
History.state = notready
state 1.2:
[executing process Move_Forward]
state 1.3:
[executing process Move_Forward]
Move_Forward.event = Null
Move_Forward.state = ready
state 1.4:
[executing process Move_Forward]
Move_Forward.event = Activate
state 1.5:
[executing process Move_Back]
Move_Forward.state = active
History.event = Activate
state 1.6:
[executing process Move_Back]
Move_Back.event = Null
Move_Back.state = ready
state 1.7:
[executing process Move_Back]
Move_Back.event = Activate
state 1.8:
Move_Back.state = active
```

Figure 2: SMV counterexample for the activity announcement solution of the *History* hierarchy.

are recursive with respect to MODULE instantiation which allows one to compactly define modules that can *peek* at the states of arbitrarily many machines. The actual state duplication to support this peeking is done at MODULE instantiation time. The expression is powerful and particularly useful for quantifying over the behavior of all machines in an inductive structure like a task hierarchy. As with any powerful feature, it must be applied symmetrically and methodically.

Machines are extended with the DEFINE macro `BUSY` that summarizes activity in the hierarchy rooted by the machine. For leaf machines, the declaration is:

```
BUSY := (state = active)
```

whereas for machines like **alt** that represent internal nodes in hierarchies, the `BUSY` macro includes the `BUSY` for each subtask:

```
BUSY := (state = active) | child1.BUSY | child2.BUSY
```

With this macro defined for each class of machine, other machines can compactly ascertain the activity of all machines in a sub–hierarchy by referring to the `BUSY` macro of the machine that tops the hierarchy. Ordering task machines are extended with two macros: `PAREXCL1` and `PAREXCL2` that encapsulate exclusion criteria induced by parent machines. Exclusion criteria are built up from logical conditions about the `BUSY`–ness of sibling hierarchies according to parent tasks. For alternative machines, the exclusion criteria for the first subtask machine (`PAREXCL1`) states that the second subtask machine is not experiencing activity at any level in its hierarchy. If the first subtask machine could somehow access this knowledge, it could use in as a precondition for becoming active.

By supplying these exclusion criteria as MODULE parameters, **leaf** machines will have access to the exclusion appropriate to their level in the hierarchy. The leaf transition to the `active` state can then have the additional condition that there is no sibling activity that would preclude this machine going `active`. The peeking declarations for **leaf** and **alt** appear in the appendix. SMV confirms that this design upholds the safety specification for the *History* example.

5 Addressing Design Tradeoffs

The iterative approach addresses correctness requirements but does not directly address efficiency requirements. Typical specification languages might have a collection of correct operationalizations. Some of these will have more desirable non–functional properties than others. Ideally, a mechanized design process converges to a solution that is both correct and efficient.

5.1 An Example Tradeoff

The tradeoff between peeking and announcement is illustrative. Peeking at the internal state of concurrent machines requires support from the run–time interpreter. When the degree of peeking is excessive, this support can be prohibitively expensive. The SMV macros we use to implement peeking expand into an amount of state proportional to the depth of a leaf node in a hierarchy. In the worst case, this expansion is exponential. The alternative to intrusive peeking is to communicate control by issuing events. The an-

nouncement based design failed because control was delocalized and sometimes machines could carry out illegal behavior before the control events disabling such behavior had time to arrive. Though the approach failed to hold up under correctness scrutiny, it has very desirable efficiency properties. The two approaches are at opposite end points of a design spectrum. Perhaps a hybrid design exists that has the correctness properties of the peeking solution and the efficiency properties of the announcement solution.

What makes the announcement solution fail is that it assumes too much about the relative scheduling of concurrent machines. The model checker looks at all possible interleavings to make sure that designs are not sensitive to such orderings. Delivered interpreters, however, will only simulate parallel execution on a sequential machine. If the scheduling policy is fixed by the design, the engineer could use knowledge of the schedule when designing state machines to reduce the amount of duplicated state. Conversely, if the engineer determines that a scheduling assumption enables an efficient state machine design, he could fix the interpreter scheduling policy to be one that reflects that assumption. This adds another degree of freedom to the design process.

For example, we believe a prioritized scheduling policy enables an efficient compromise between the peeking and event announcement solutions. If machines only need to peek at parent states to guard themselves from going active, the amount of replicated state would be linear in the depth of the hierarchy as opposed to exponential. This is safe as long as events issued to parent machines reach those machines before other machines that the parent controls can execute. Since `Activate` events propagate up the hierarchy, we believe that a prioritized scheduling algorithm in concert with parent peeking would uphold the safety semantics of the ordering operators.

5.2 Automated Design Support

This opens an interesting question regarding the automation of interpreter design: Can we extend the model checking framework to enable engineers to experiment with tradeoffs of state machine design and scheduling policy? Extension requires a suitable metaphor for scheduling policy in terms understood by model checking technology.

Scheduling policies can be thought of as constraints on the sequences of feasible machine executions. The SMV model checker associates a boolean value `running` with each state machine. The `running` value is true when the associated machine is executing in that state and false otherwise. With the interleaved process model of execution, only one machine is ever running in any given state. The `running` value is treated as just another state variable and can be referred to in CTL expressions. This allows engineers to express execution ordering constraints—scheduling policies—using CTL. Conceptually, these expressions represent path filters, and invariant specifications should only be applied to paths that are not filtered out by these expressions. The feasibility of reasoning about design tradeoffs between state complexity and scheduling policy now rests on two assumptions:

1. CTL is a comfortable mechanism for denoting the scheduling policies engineers want to reason about, and

2. The path filter architecture has a realization either through some CTL construct or some explicit facility in a given model checker.

5.3 Difficulties Using CTL

We believe that CTL is a comfortable schedule specification mechanism. Consider the proposed prioritized scheduling policy acting on the WWW browser history interaction example. All paths that are consistent with this policy will uphold the following invariant at every point along those paths:

```
((Move_Forward.running -> (!(EX History.running) |
                           (AX History.running))) &
 (Move_Back.running    -> (!(EX History.running) |
                           (AX History.running))))
```

This says that if `Move_Forward` or `Move_Back` is running in the current state, then either `History` cannot run in the next state or it necessarily runs in the next state. The condition can be synthesized for an arbitrary task hierarchy.

We first attempted to bring scheduling reasoning into our framework by *guarding* the correctness invariant specifications with these schedule invariants. The general form of correctness invariant is:

```
AG (scheduling-invariant
    ->
    correctness-invariant)
```

Unfortunately, this form admits paths for which the `scheduling-invariant` does not hold globally. Consider a path $p_1 p_2 \ldots p_k q_1 q_2 \ldots$. The p_i states are such that the scheduling invariant is not true, and the q_i states are such that it is true. When the antecedent of an implication is false, the implication is true. This path would be admissible according to the specification. Scheduling invariants cannot be combined with correctness invariants in this manner because too many paths are admissible.

A mechanism is needed that will apply the correctness invariant only to paths for which the scheduling invariant holds globally. Functionality for this *path selection* mechanism must be provided by the model checker. SMV takes steps in this direction. In SMV, specifications are only quantified over what it calls *fair* paths. Fair paths are specified with `FAIRNESS` constraints of the form:

```
FAIRNESS
    ctl-form
```

where `ctl-form` is any CTL expression. SMV determines paths for which these constraints are true "infinitely often" and uses only these paths when checking specifications.

Unfortunately, our problem dictates the need to declare `ctl-form` to be true globally over all fair paths. Without this ability, we cannot use SMV as the enabling technology for our approach. However, if SMV could be extended to restrict its attention to paths for which a given CTL expression holds globally, we believe that our framework will support design tradeoffs between state machine complexity and scheduling policy.

6 Conclusion

We have demonstrated the utility of model checking in supporting the operationalization of a declarative specification language. By providing support to check correctness, the model checker frees engineers to explore design tradeoffs between competing non-functional requirements. In the MASTERMIND example, the primary non-functional requirement was efficient interpretation. The optimal solution is a compromise between scheduling policy and state space representation. The formalization of this compromise is easily expressed in second order CTL. Some model checkers provide limited second order capabilities (such as SMV's `FAIRNESS` constraints). Our framework suggests the usefulness of more general second order facilities in model checking technology.

References

[1] Gregory D. Abowd, Hung-Ming Wang, and Andrew F. Monk. A formal techniqe for automated dialogue development. In Gary M. Olson and Sue Schuon, editors, *Proceedings of the Symposium on Designing Interactive Systems: Processes, Practices, Methods & Techniques*, pages 219–226. ACM Press, August 1995.

[2] Joanne Atlee and John Gannon. State-based model checking of event driven systems requirements. *IEEE Transactions on Software Engineering*, 19(3), January 1993.

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the* 5^{th} *International Symposium on Logic in Computer Science*, June 1990.

[4] Alan Dix, Janet Finlay, Gergory Abowd, and Russell Beale. *Human-Computer Interaction*. Prentice Hall, New York, 1993.

[5] Nicolas Halbwachs. About synchronous programming and abstract interpretation. In *First International Static Analysis Symposium, SAS'94*, pages 179–192, 1994.

[6] David Harel. On visual formalisms. *Communications of the ACM*, 31(5), 1988.

[7] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Publishing Company, 1979.

[8] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In *International Symposium on Software Testing and Analysis (ISSTA'96)*, 1996.

[9] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992. CMU-CS-92-131.

[10] R. Neches, J. Foley, P. Szekely, P. Sukaviriya, P. Luo, S. Kovacevic, and S. Hudson. Knowledgeable development environments using shared design models. In *Intelligent Interfaces Workshop*, pages 63–70, January 1993.

[11] Pedro Szekely, Ping Luo, and Robert Neches. Beyond interface builders: Model-based interface tools. In *Human Factors in Computing Systems — INTERCHI'93*, pages 383–390. Addison Wesley, April 1993.

[12] Jeannette M. Wing and Mandana Vaziri-Farahani. Model checking software systems: A case study. In *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995.

Appendix

This is the complete specification for the peeking solution with an instantiation for the WWW browser *History* task hierarchy.

The MODULE **leaf** represents an interaction task.

```
MODULE leaf(parent,parevent, pexcl)
VAR
   event    : Null, Disable, Enable, Activate;
   state    : notready, ready, active;
ASSIGN
   init(state) := notready;
   init(event) := Null;
   next(state) :=
        case
             (state = notready) &
              (event = Enable)    : ready;
             (state = ready) &
              (event = Disable)   : notready;
             (state = ready) &
              !pexcl &
               (event = Activate) : active;
             state = active       : notready;
             1                           : state;
        esac;
   next(event) :=
        case
             (state = ready) &
              !pexcl        : Activate, Null;
             1                    : Null;
        esac;
   next(parevent) :=
        case
            (state = ready) &
             !pexcl &
              (event = Activate)   : Activate;
            1                      : parevent;
        esac;
DEFINE
   BUSY := (state = active);
```

The MODULE **alt** enforces an alternative decomposition between two subtask machines:

```
MODULE alt(parent, pevent, pexcl,
          child1, c1event,
          child2, c2event)
VAR
   event      : Null, Disable, Enable, Activate;
   state      : notready, ready, active;
ASSIGN
   init(state)     := notready;
   init(event)     := Null;
   next(state)     :=
        case
            (state = notready) &
             (event = Enable) : ready;
            (state = ready) &
             (event = Disable) : notready;
            (state = ready) &
             (event = Activate) : active;
            (state = active) &
             (child1.state = notready) &
              (child2.state = notready) : notready;
            1 : state;
        esac;
   next(c1event)   :=
        case
            (next(state) = ready) : Enable;
            (next(state) = notready) : Disable;
            (next(state) = active) &
             (child1.state = ready) : Disable;
            1 : c1event;
        esac;
   next(c2event)   :=
        case
            (next(state) = ready) : Enable;
            (next(state) = notready) : Disable;
            (next(state) = active) &
             (child2.state = ready) : Disable;
            1 : c2event;
        esac;
   next(pevent)    :=
        case
            (state = ready) &
             (next(state) = active) : Activate;
            1 : pevent;
        esac;
DEFINE
   BUSY      := (state = active) |
                child1.BUSY |
                child2.BUSY;
   PAREXCL1  := (pexcl | (child2.BUSY));
   PAREXCL2  := (pexcl | (child1.BUSY));
```

Since machines compose hierarchically with connections to parent machines, we need a machine with no parent to be the **top** of the hierarchy:

```
MODULE top(cevent)
VAR
   event      : Null, Disable, Enable, Activate;
   state      : notready,ready, active;
ASSIGN
   init(state) := notready;
   init(event) := Enable;
   next(cevent) :=
        case
           (state = ready) : Enable;
           1               : cevent;
        esac;
   next(state) :=
        case
            (state = notready) &
             (event = Enable) : ready;
            (state = ready) &
             (event = Activate) : active;
            1 : state;
        esac;
   next(event) := Null;
DEFINE
   PAREXCL   := (state = active);
```

This MODULE main instantiates the *History* hierarchy and the correctness invariants for alternative tasks:

```
MODULE main
VAR
   Move_Forward  : process
                   leaf(History,
                        History.event,
                        History.PAREXCL1);
   Move_Back     : process
                   leaf(History,
                        History.event,
                        History.PAREXCL2);
   History : process
             alt(root, root.event, root.PAREXCL,
                 Move_Forward,
                 Move_Forward.event,
                 Move_Back,
                 Move_Back.event);
   root        : process
                 top(History.event);
SPEC
   AG !((Move_Forward.state = active) &
        (Move_Back.state = active))
SPEC
   EF((Move_Back.state = ready) &
      (Move_Forward.state = ready))
```